Recurrent Neural Networks Trading Strategy | LSTM

aiquants Research | © Dayton Nyamai

Application of RNNs in Market Direction Prediction

Summary

This project aims to leverage machine learning techniques, specifically Recurrent Neural Networks (RNNs) to develop a trading strategy that can generate consistent profits in the financial markets. By utilizing historical data and advanced algorithms, we aim to identify patterns and trends that can be exploited for profitable trading opportunities.

[147]:
```python
# Import the necessary libraries
import pandas as pd
import numpy as np
import datetime as dt
import time
```

[148]:
```python
from pylab import mpl, plt
plt.style.use('seaborn-v0_8')
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

The Data

[149]:
```python
# Load the historical data
raw = pd.read_csv('EURUSD_5M_data.csv', index_col=0, parse_dates=True).dropna()
raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5688 entries, 2023-07-03 00:00:00 to 2023-07-28 22:25:00
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Open    5688 non-null   float64
 1   High    5688 non-null   float64
 2   Low     5688 non-null   float64
 3   Close   5688 non-null   float64
dtypes: float64(4)
memory usage: 222.2 KB
```

[150]:
```
raw.head()
```

[150]:
```
                        Open     High      Low    Close
Datetime
2023-07-03 00:00:00  1.09123  1.09123  1.09111  1.09111
2023-07-03 00:05:00  1.09123  1.09123  1.09123  1.09123
2023-07-03 00:10:00  1.09135  1.09135  1.09123  1.09123
2023-07-03 00:15:00  1.09123  1.09123  1.09123  1.09123
2023-07-03 00:20:00  1.09123  1.09123  1.09123  1.09123
```

Calculates the average proportional transactions costs

[151]:
```
# Specify the average bid-ask spread.
spread = 0.0002

# Calculate the mean closing price
mean = raw['Close'].mean()

ptc = spread / mean
ptc.round(6)
```

[151]:
```
0.000181
```

Calculate log returns and create direction column

[152]:
```
data = pd.DataFrame(raw['Close'])
data.rename(columns={'Close': 'price'}, inplace=True)

data['returns'] = np.log(data['price'] / data['price'].shift(1))
data.dropna(inplace=True)
data['direction'] = np.where(data['returns'] > 0, 1, 0)
data.round(6).head()
```

[152]:
```
                       price  returns  direction
Datetime
2023-07-03 00:05:00  1.09123  0.00011          1
2023-07-03 00:10:00  1.09123  0.00000          0
2023-07-03 00:15:00  1.09123  0.00000          0
2023-07-03 00:20:00  1.09123  0.00000          0
2023-07-03 00:25:00  1.09111 -0.00011          0
```

A histogram providing visual representation of the EUR log returns distribution

[153]:
```
data['returns'].hist(bins=35, figsize=(10, 6));
# Add figure caption
plt.figtext(0.5, -0.01, 'Fig. 1.1 A histogram  showing the distribution of EUR␣
↪log returns ', style='italic',ha='center')

# Show the plot
plt.show()
```

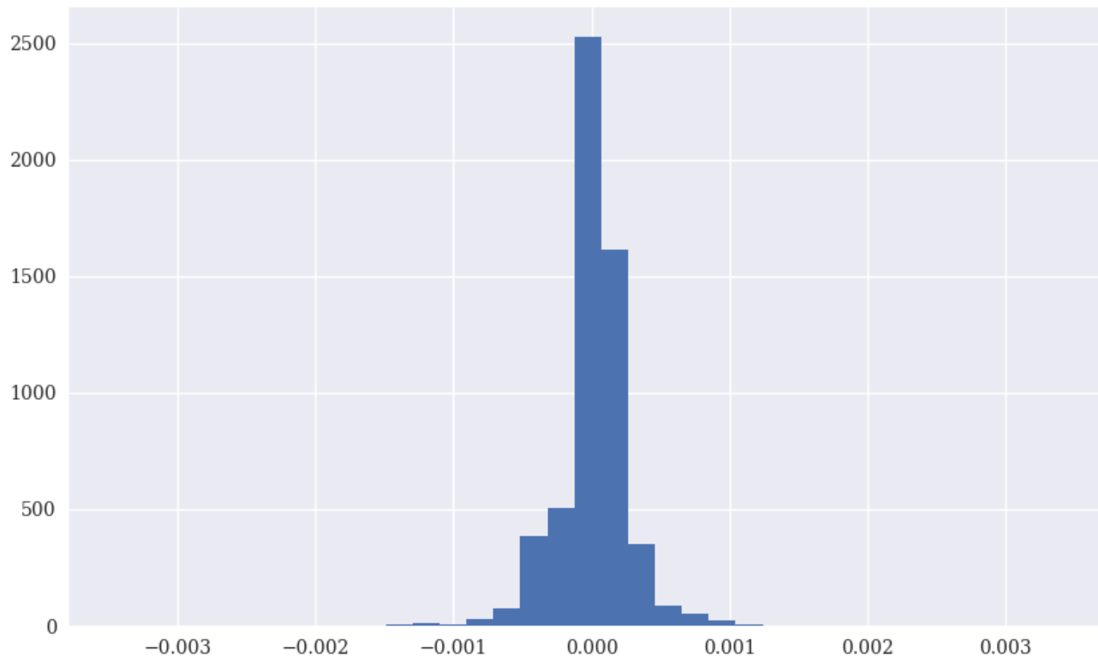

Fig. 1.1 A histogram showing the distribution of EUR log returns

Second, create the features data by lagging the log returns and visualize it in combination with the returns data. We can use various visualization techniques such as scatter plots or line plots to compare the lagged log returns with the returns data.

Create lagged columns

[154]:
```
lags = 5

cols =[ ]
for lag in range(1, lags+1):
    col =  f'lag_{lag}'
    data[col] = data['returns'].shift(lag)
    cols.append(col)
data.dropna(inplace=True)
```

```
data.round(6).tail()
```

[154]:

```
                       price  returns  direction     lag_1     lag_2  \
Datetime
2023-07-28 22:05:00  1.10193      0.0          0 -0.000109  0.000109
2023-07-28 22:10:00  1.10193      0.0          0  0.000000 -0.000109
2023-07-28 22:15:00  1.10193      0.0          0  0.000000  0.000000
2023-07-28 22:20:00  1.10193      0.0          0  0.000000  0.000000
2023-07-28 22:25:00  1.10193      0.0          0  0.000000  0.000000

                        lag_3     lag_4     lag_5
Datetime
2023-07-28 22:05:00  0.000000  0.000000 -0.000327
2023-07-28 22:10:00  0.000109  0.000000  0.000000
2023-07-28 22:15:00 -0.000109  0.000109  0.000000
2023-07-28 22:20:00  0.000000 -0.000109  0.000109
2023-07-28 22:25:00  0.000000  0.000000 -0.000109
```

Scatter plot based on features and labels data

[155]:

```
data.plot.scatter(x='lag_1', y='lag_2', c='returns',
                  cmap='coolwarm', figsize=(10, 6), colorbar=True)

# Adding vertical and horizontal lines at 0
plt.axvline(0, c='r', ls='--')
plt.axhline(0, c='r', ls='--')

# Add figure caption
plt.figtext(0.4, -0.03,
            'Fig. 1.2 A scatter plot based on features and labels data',
            style='italic',ha='center')

# Show the plot
plt.show()
```

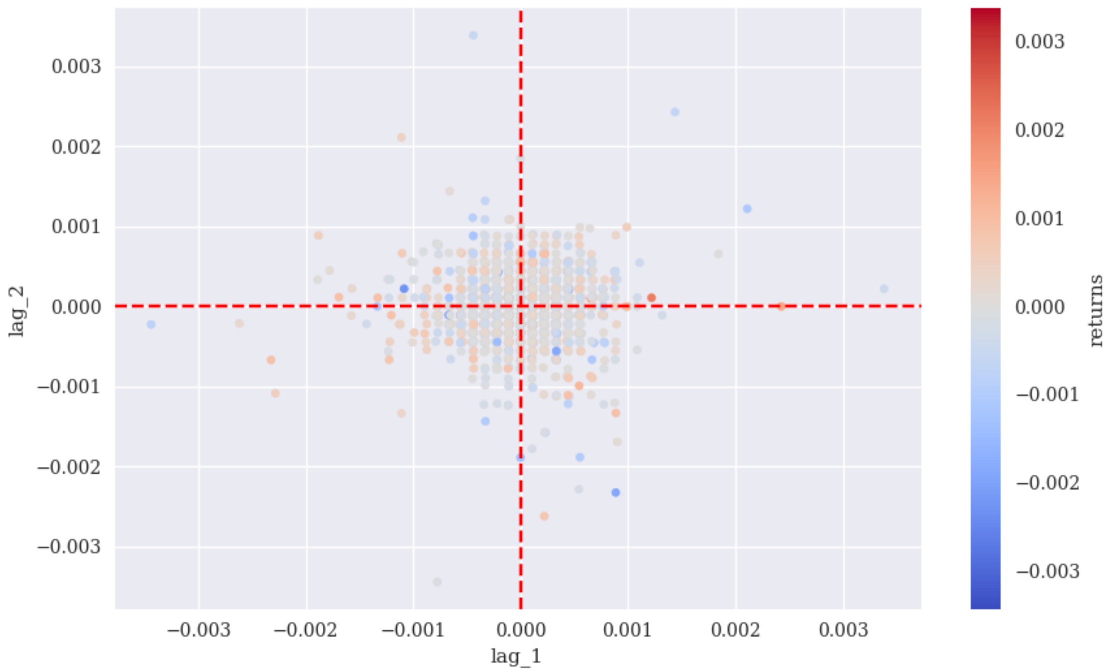

Fig. 1.2 A scatter plot based on features and labels data

With the dataset fully prepared, various deep learning techniques can be employed to forecast market movements based on the provided features. Additionally, these predictions can be utilized to rigorously backtest a trading strategy.

Deep Learning Models: Recurrent Neural Networks

Recurrent Neural Networks (RNNs): There are several types of RNNs that are well-suited for sequence prediction tasks. In our case, we will explore the **Long Short-Term Memory (LSTM)**.

Summary

In this task, we will create an LSTM (Long Short-Term Memory) model for predicting future market movements. We will also utilize the TPU (Tensor Processing Unit) VM cloud infrastructure from Google, for efficient training and inference.

Import the necessary libraries, tensorFlow and its submodules

[156]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from sklearn.preprocessing import StandardScaler
#from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
```

Split the data into training and test sets

[157]:
```python
# Split the data into training and test sets
split = int(len(data) * 0.80)
training_data =  data.iloc[:split].copy()
test_data = data.iloc[split:].copy()
```

Standardize the training and test data.

[158]:
```python
mu, std = training_data.mean(), training_data.std()
training_data_ = (training_data - mu) / std
test_data_ = (test_data - mu) / std
```

Reshape the training and test data for LSTM input

[159]:
```python
X_train = np.array(training_data_[cols])
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
y_train = np.array(training_data['direction'])

X_test = np.array(test_data_[cols])
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
y_test = np.array(test_data['direction'])
```

Build the LSTM model

[160]:
```python
model = Sequential()
model.add(LSTM(128, activation='relu', input_shape=(lags, 1)))
model.add(Dense(1, activation='sigmoid'))
```

Compile the model

[161]:
```python
optimizer = Adam(learning_rate=0.0001)
model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

Train the model

[162]:
```python
model.fit(X_train, y_train, epochs=100, verbose=False, validation_split=0.2,␣
 ↪shuffle=False)

res = pd.DataFrame(model.history.history)
res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
plt.figtext(0.5, 0.02, 'Fig. 1.3 Accuracy of the LSTM model on training and␣
 ↪validation data per training step', style='italic',ha='center')
plt.show()
```

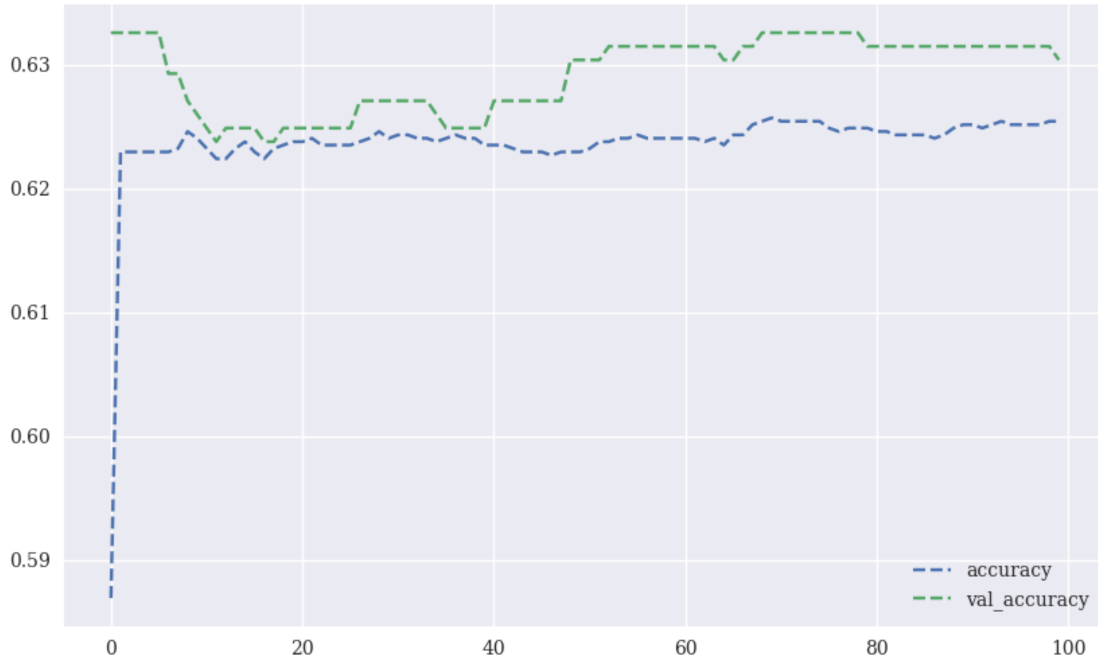

Fig. 1.3 Accuracy of the LSTM model on training and validation data per training step

Evaluate the performance of the model on training data

[163]:
```
train_loss, train_accuracy = model.evaluate(X_train, y_train)
```

```
143/143 [==============================] - 0s 2ms/step - loss: 0.6555 -
accuracy: 0.6268
```

Training Data

Make predictions on the training data

[164]:
```
train_predictions = np.where(model.predict(X_train) > 0.5, 1, 0)
```

```
143/143 [==============================] - 0s 2ms/step
```

Transforms the predictions into long-short positions, +1 and -1

[165]:
```
training_data['prediction'] = np.where(train_predictions > 0, 1, -1)
```

The number of the resulting short and long positions, respectively.

[166]:
```
training_data['prediction'].value_counts()
```

[166]:
```
-1    4358
 1     187
Name: prediction, dtype: int64
```

Trading Rules

In the benchmark case, i.e training_data['returns'], we adopt a long position on the asset throughout the entire period. This means that we hold one unit of the asset for the entire duration. On the other hand, in the case of the DNNs strategy, i.e training_data['strategy'], we take either a long or short position on the asset, i.e one unit of the asset.

Calculates the strategy returns given the positions

[167]:
```python
training_data['strategy'] = training_data['prediction'] * training_data['returns']
training_data[['returns', 'strategy']].sum().apply(np.exp)
```

[167]:
```
returns     1.014389
strategy    1.002605
dtype: float64
```

Plots and compares the strategy performance to the benchmark performance (in-sample)

[168]:
```python
training_data[['returns', 'strategy']].cumsum().apply(np.exp).plot(figsize=(10, 6));
plt.figtext(0.5, 0.02, 'Fig. 1.4 Gross performance of EUR/USD compared to the ML-based strategy', style='italic',ha='center')
plt.figtext(0.5, -0.03, '(in-sample, no transaction costs)', style='italic',ha='center')

plt.show()
```

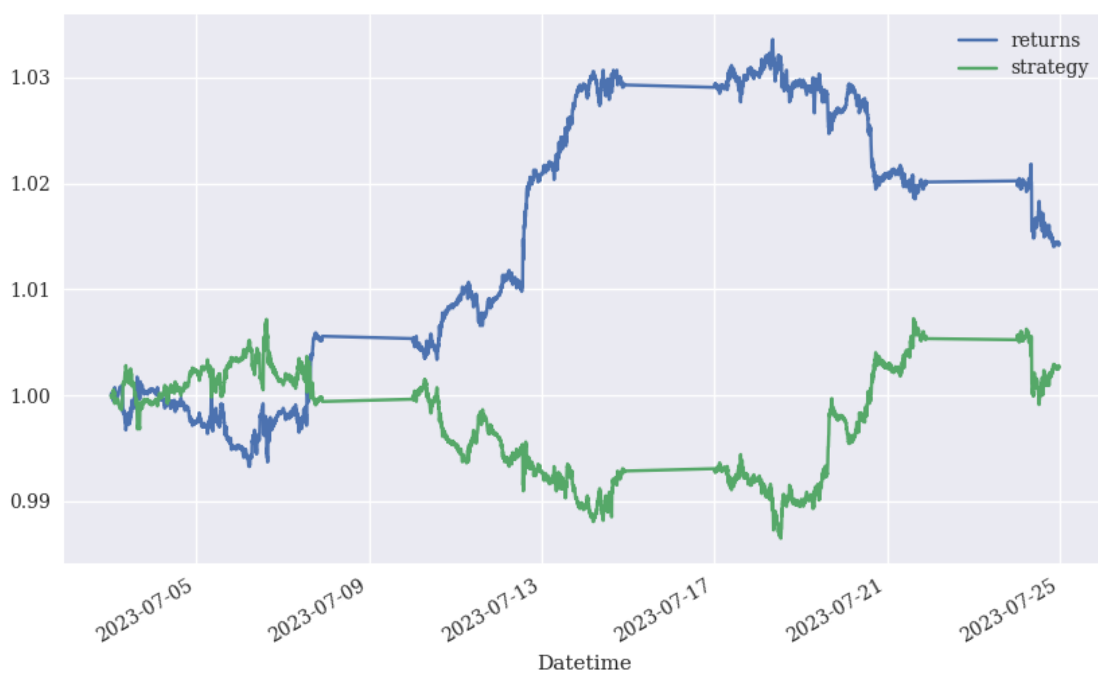

Fig. 1.4 Gross performance of EUR/USD compared to the ML-based strategy

(in-sample, no transaction costs)

Testing Data

Evaluate the performance of the model on testing data

[169]:
```
model.evaluate(X_test, y_test)
```

```
36/36 [==============================] - 0s 2ms/step - loss: 0.6643 - accuracy:
0.6157
```

[169]:
```
[0.6643161177635193, 0.615655243396759]
```

Make predictions on the test data

[170]:
```
test_predictions = np.where(model.predict(X_test) > 0.5, 1, 0)

# Transforms the predictions into long-short positions, +1 and -1
test_data['prediction'] = np.where(test_predictions > 0, 1, -1)
```

```
36/36 [==============================] - 0s 2ms/step
```

The number of the resulting short and long positions, respectively.

[171]:
```
test_data['prediction'].value_counts()
```

[171]:
```
-1    1075
 1      62
Name: prediction, dtype: int64
```

Calculate the strategy returns given the positions, with the proportional transaction costs included

[172]:
```
test_data['strategy'] = test_data['prediction'] * test_data['returns']

test_data['strategy_tc'] = np.where(test_data['prediction'].diff() != 0,
                                    test_data['strategy'] - ptc,␣
 ↪test_data['strategy'])

test_data[['returns', 'strategy', 'strategy_tc']].sum().apply(np.exp) # ␣
 ↪strategy_tc: with the proportional transaction costs
```

[172]:
```
returns        0.995591
strategy       0.986798
strategy_tc    0.977741
dtype: float64
```

Plots and compares the strategy performance to the benchmark performance (out-of-sample)

[173]:
```
test_data[['returns', 'strategy', 'strategy_tc']].cumsum().apply(np.exp).
  ↪plot(figsize=(10, 6));
plt.figtext(0.5, -0.06, 'Fig. 1.5 Performance of EUR/USD exchange rate and␣
  ↪ML-based algorithmic trading strategy', style='italic',ha='center')
plt.figtext(0.5, -0.1, '(out-of-sample, with transaction costs)',␣
  ↪style='italic',ha='center')

plt.show()
```

Fig. 1.5 Performance of EUR/USD exchange rate and ML-based algorithmic trading strategy (out-of-sample, with transaction costs)

Optimal Leverage

Equipped with the trading strategy's log returns data, the mean and variance values can be calculated in order to derive the optimal leverage according to the Kelly criterion.

Annualized mean returns

[174]:
```
mean = test_data[['returns', 'strategy_tc']].mean() * len(data) *12
mean
```

[174]:
```
returns        -0.264989
strategy_tc    -1.349938
dtype: float64
```

Annualized variances

[175]:
```
var = test_data[['returns', 'strategy_tc']].var() * len(data) * 12
var
```

[175]:
```
returns        0.006241
strategy_tc    0.006316
dtype: float64
```

Annualized volatilities

[176]:
```
vol = var ** 0.5
vol
```

[176]:
```
returns        0.079000
strategy_tc    0.079472
dtype: float64
```

Optimal leverage according to the Kelly criterion ("full Kelly")

[177]:
```
mean / var
```

[177]:
```
returns         -42.458970
strategy_tc    -213.738274
dtype: float64
```

Optimal leverage according to the Kelly criterion ("half Kelly")

[178]:
```
 mean / var * 0.5
```

[178]:
```
returns         -21.229485
strategy_tc    -106.869137
dtype: float64
```

Using the "half Kelly" criterion, the optimal leverage for the trading strategy is about 30. The graph below shows in comparison the performance of the trading strategy with transaction costs for different leverage values

Scales the strategy returns for different leverage values

[179]:
```
to_plot = ['returns', 'strategy_tc']
for lev in [10, 15, 30, 40, 50]:
    label = 'lstrategy_tc_%d' % lev
    test_data[label] = test_data['strategy_tc'] * lev
    to_plot.append(label)
```

```
test_data[to_plot].cumsum().apply(np.exp).plot(figsize=(10, 6));
plt.figtext(0.5, -0.06, 'Fig. 1.6 Performance of ML-based trading strategy for␣
  ↪different leverage values', style='italic',ha='center')
plt.show()
```

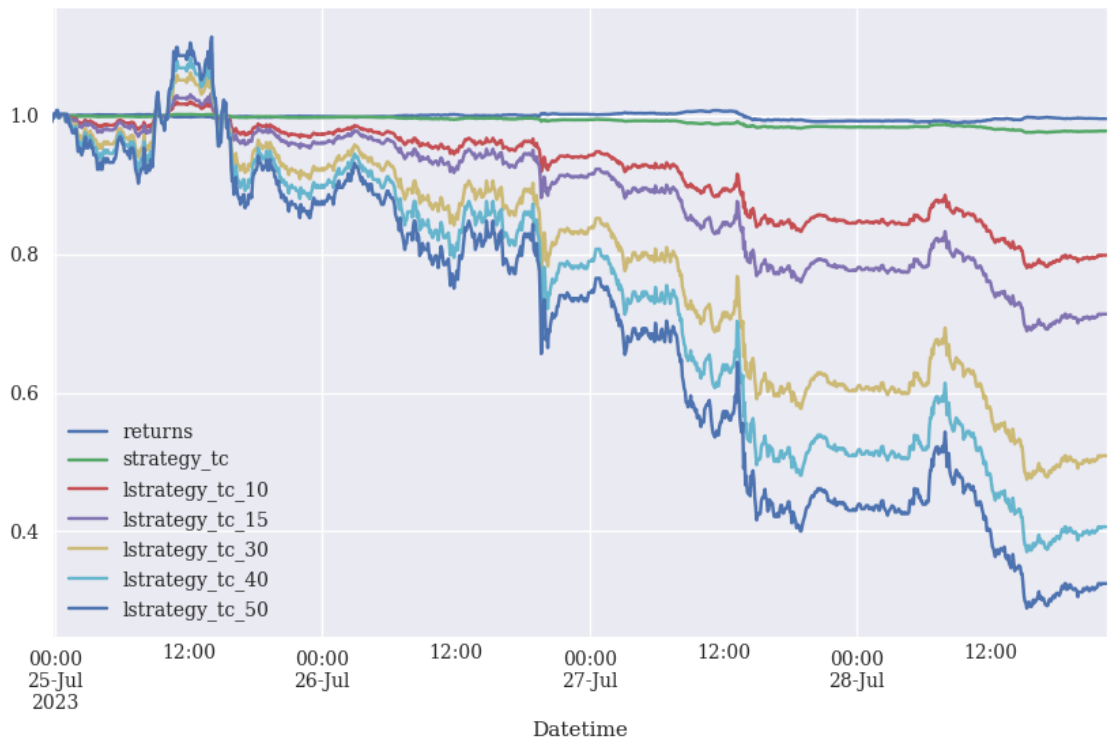

Fig. 1.6 Performance of ML-based trading strategy for different leverage values

Risk Analysis

Since leverage increases the risk associated with a trading strategy, a more in-depth risk analysis seems in order. This risk analysis involves calculating the maximum drawdown, which represents the largest loss experienced after a recent high, and the longest drawdown period, which is the duration it takes for the strategy to recover to a recent high. The analysis assumes an initial equity position of 3,333 EUR, resulting in a position size of 100,000 EUR with a leverage ratio of 30. It also assumes that there are no adjustments made to the equity over time, regardless of the strategy's performance.

Initial equity

[180]:
```
equity = 3333
```

The relevant log returns time series

[181]:
```
risk = pd.DataFrame(test_data['lstrategy_tc_30'])
```

Scaled by the initial equity

[182]:
```
risk['equity'] = risk['lstrategy_tc_30'].cumsum(
                 ).apply(np.exp) * equity
```

The cumulative maximum values over time

[183]:
```
risk['cummax'] = risk['equity'].cummax()
```

The drawdown values over time

[184]:
```
risk['drawdown'] = risk['cummax'] - risk['equity']
```

The maximum drawdown value

[185]:
```
risk['drawdown'].max()
```

[185]:
```
1973.4368497555515
```

The point in time when it happens

[186]:
```
t_max = risk['drawdown'].idxmax()
t_max
```

[186]:
```
Timestamp('2023-07-28 15:15:00')
```

Technically a new high is characterized by a drawdown value of 0. The drawdown period is the time between two such highs. The figure below visualizes both the maximum drawdown and the drawdown periods:

[187]:
```
temp = risk['drawdown'][risk['drawdown'] == 0]

periods = (temp.index[1:].to_pydatetime() -
           temp.index[:-1].to_pydatetime())

periods[20:30]
```

[187]:
```
array([], dtype=object)
```

[188]:
```
t_per = periods.max()
t_per
```

[188]:
```
datetime.timedelta(seconds=31800)
```

[189]:
```
risk[['equity', 'cummax']].plot(figsize=(10, 6))
plt.axvline(t_max, c='r', alpha=0.5);

plt.figtext(0.5, -0.06, 'Fig. 1.7 Maximum drawdown (vertical line) and drawdown␣
  ↪periods (horizontal lines)', style='italic',ha='center')
plt.show()
```

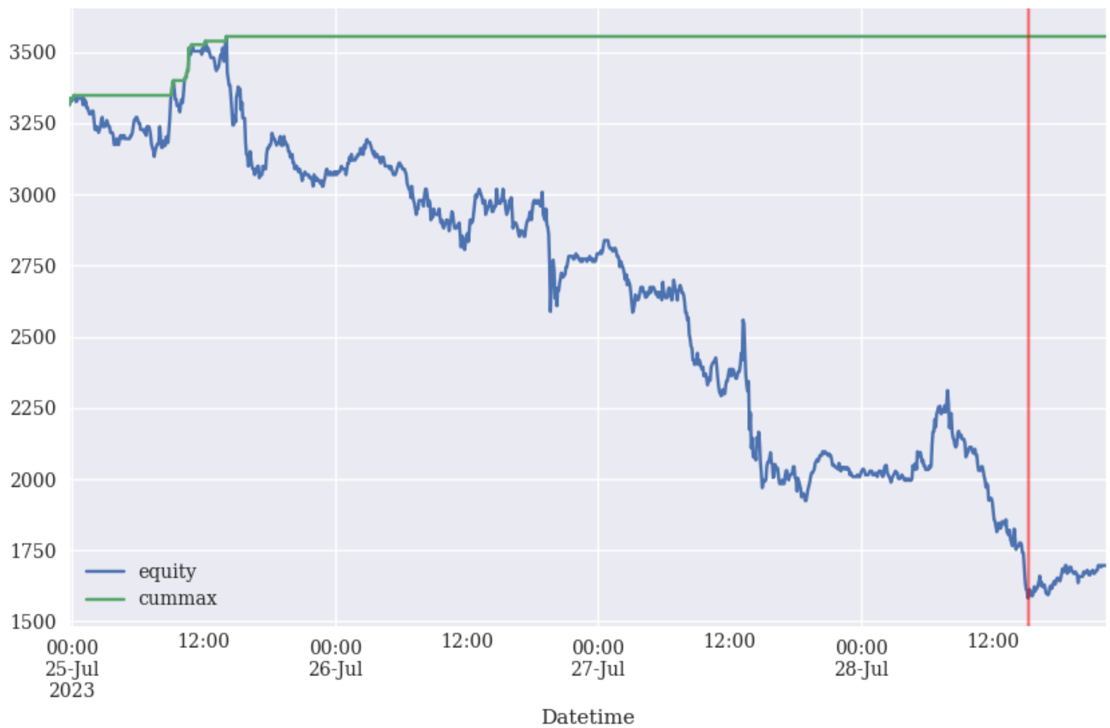

Fig. 1.7 Maximum drawdown (vertical line) and drawdown periods (horizontal lines)

Value-at-Risk (VaR)

VaR, quoted as a currency amount represents the maximum loss to be expected given both a certain time horizon and a confidence level. The code that follows derives VaR values based on the log returns of the equity position for the leveraged trading strategy over time for different confidence levels. The time interval is fixed to the bar length of 5 min:

Defines the percentile values to be used

[190]:
```
import scipy.stats as scs

percs = np.array([0.01, 0.1, 1., 2.5, 5.0, 10.0])
risk['returns'] = np.log(risk['equity'] /
                         risk['equity'].shift(1))

# Calculate the VaR values given the percentile values
VaR = scs.scoreatpercentile(equity * risk['returns'], percs)

def print_var():
    print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
    print(33 * '-')
    for pair in zip(percs, VaR):
        print('%16.2f %16.3f' % (100 - pair[0], -pair[1]))
```

```
# Translate the percentile values into confidence levels and the VaR values
print_var()
```

```
Confidence Level    Value-at-Risk
---------------------------------
           99.99          245.825
           99.90          207.864
           99.00           98.932
           97.50           66.298
           95.00           46.186
           90.00           33.205
```

VaR values for a time horizon

Time horizon: 1 hour

Resample the original DataFrame object, in effect the VaR values are increased for all confidence levels

[191]:
```
# Resample the data from five-minute to one-hour bars.
hourly = risk.resample('1H', label='right').last()
```

[192]:
```
hourly['returns'] = np.log(hourly['equity'] /
                           hourly['equity'].shift(1))
```

[193]:
```
# Recalculates the VaR values for the resampled data.
VaR = scs.scoreatpercentile(equity * hourly['returns'], percs)
print_var()
```

```
Confidence Level    Value-at-Risk
---------------------------------
           99.99          328.835
           99.90          324.252
           99.00          278.419
           97.50          246.211
           95.00          211.337
           90.00          153.690
```

NEXT: Persisting the Model Object